LOTEC: A Simple DSM Consistency Protocol for Nested Object Transactions

Peter Graham* and Yahong Sui
Parallel and Distributed Systems Lab
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba
Canada R3T 2N2

Abstract

In this paper, we describe an efficient software-only Distributed Shared Memory (DSM) consistency protocol for an unconventional but important application domain - object transactional processing. Designers of transactional applications, while sensitive to performance issues, already accept significant overhead for the added functionality provided by transactions and, we speculate, would be willing to accept some small additional overhead for the benefit of "easy" distributed parallel execution. This is in contrast to the performance *critical* needs of the high-end scientific applications typically targetted by DSM system designers. While, after 10 years of performance enhancement, software DSMs are still considered inadequate by a large part of the parallel computing community, we believe they are now more than sufficiently mature to support distributed transactional environments.

The key to making DSM attractive to the developers of transactional applications is ease of use. The much touted simplification offered by the use of shared memory instead of message passing is not enough. It must be possible to hide the needed synchronization operations from the programmer. To achieve this goal, we have developed a system for nested object transactions that supports the automatic insertion of synchronization primitives. These primitives are then used to drive the operation of an improved form of Bershad's Entry Consistency (EC) protocol to maintain memory consistency between the processors/nodes in a distributed computing system. The protocol described is also compatible with a number of performance enhancements for DSM systems that have been described in the literature.

1 Introduction and Related Work

Over the last dozen years, the development of software-only Distributed Shared Memory (DSM) systems has been widely investigated. Following the pioneering work of Li and Hudak [LH86] a large body of knowledge has been created focusing, primarily, on improving the performance of such systems by reducing the number and size of messages sent. This has been accomplished using weak consistency protocols [CBZ91, KCZ92, BZS93] as well as other optimizations [CBZ95, ACDZ97, ACRZ97, Lu97, SB98, BPA98]. Work has also been done on decreasing the overhead of the transmission protocol used to send the required consistency maintenance messages [vECGS92, vEBBV95, Buo98] and on the underlying networks themselves (e.g. Gigabit Ethernet, Myrinet [BCF+95]). Despite this, the performance of software DSM systems is still significantly less than many researchers in the parallel processing community would like. This is not so much a problem with the design and implementation of these systems as it is a natural tendency for the performance of new software systems to lag behind the performance of hardware (or well established software) systems.

Not all applications require the kind of raw performance with minimal overhead that, for example, Computational Fluid Dynamics problems do [DVL92]. There is a wide range of important applications that could benefit significantly from the improved performance levels already available using current DSM technology. These include a large number of commercial applications as well as many, less demanding, scientific and engineering ones. The key to making DSM practical for these applications is ease of use.

All efficient DSM systems are based on weak consistency models that reduce communication overhead by allowing cached copies of data to be temporarily inconsistent when it is known that this will not affect the correctness of execution. The implementation of such systems key on data synchronization operations in order to know when it is safe to allow inconsistency.[1] The developers of non-parallel applications, however, are not used to providing explicit synchronization operations to control concurrent access to shared data so this is an added complexity for them. It is also well known that the wide range of potential failure modes possible when using distributed systems further complicates the programming of distributed applications. Both of these problems can be addressed using a suitable, low-overhead transaction facility.

We have designed a programming system that uses objects as its fundamental abstraction mechanism and which provides access to them in a distributed environment via DSM. To ensure simplicity of use, a transactional execution

*This research was supported in part by the Natural Sciences and Engineering Research Council of Canada under grant OGP-0194227.

[1] Generally speaking it is safe to allow a copy of a shared datum to be inconsistent when we know it will not be accessed. For example, because a lock required to permit access to the data cannot be acquired.

model based on Moss' closed nested transactions [Mos85] is used. This ensures the correctness of concurrent executions based on the serializability of the transactions. The consistency of the DSM is then maintained using a modified form of Bershad's Entry Consistency (EC) protocol [BZS93].

Our new DSM consistency protocol, LOTEC, is both simple and efficient. It provides a form of weak consistency utilizing the synchronization operations defined by our nested *object* two-phase locking (O2PL) concurrency protocol to drive its operation. (In the same way that Release Consistency [CBZ91] and Lazy Release Consistency [KCZ92] utilize conventional lock operations to guide their consistency maintenance operations.) Further, the well-defined semantics of the objects operated on are exploited to optimize the performance of LOTEC to further reduce the amount of data transferred (relative to EC) to maintain consistency. In this paper, we present the algorithms for implementing LOTEC. We also present the results of simulation experiments that were performed to assess the effectiveness of LOTEC and to explore the desirable operating characteristics of a network environment in which it should operate.

Our work builds indirectly on earlier work on distributed object systems (including [RB94, SF95, BHAB96]) and, in some ways, on objectbase systems (including [LLOW91, Deu90]). It is most closely related to work by Fleisch and Hyde [FH98] and Itzkovitz and Schuster [IS99].

The rest of this paper is organized as follows. Section 2 describes the problem domain and motivates our solution strategy. Section 3 describes nested object transactions and our correctness criterion for them. In Section 4, we present the Lazy Object Transactional Entry Consistency (LOTEC) protocol. We then describe a simulation system we created to evaluate the effectiveness of LOTEC and to determine the range of network parameters for which it would be useful in Section 5. Finally, Section 6 concludes and discusses future directions for our research.

2 Problem Description and Motivation

Transaction processing systems are continually in need of more processing power. Unfortunately, the complexity of using parallel processing on distributed memory machines to solve such problems is great. For this reason, many transactional applications are confined to high-end uniprocessors or shared memory multiprocessors both of which are extremely costly.

Recently, there has been great interest in the use of networked *clusters* of machines to create low-cost parallel compute engines (e.g. Beowulf clusters [BTSD+95]). Such clusters typically rely on message passing programming but provide very cost-effective performance. Software-only Distributed Shared Memory (DSM) systems which provide the illusion of shared memory on virtual memory hardware have been developed in parallel but independently. Combined, these technologies (clusters and DSM) offer the potential for a practical parallel transaction processing environment.

The problem with running transactional applications on distributed memory machines is that they are not well suited to implementation using message passing. Users of transactional systems expect much of the complexity of programming to be hidden from them by the transaction mechanism. The use of user-visible message passing is inconsistent with this expectation. DSM systems, however, if properly constructed, can completely hide the required message passing from the user. Further, in a complementary way, the use of transactions hides much of the complexity of distributed programming (unexpected failure modes, etc.) from the user.

An important characteristic of transaction processing systems is that their computational requirements typically come not from the complexity of a single transaction but rather from the volume of transactions which must be concurrently processed. The design focus for transaction processing systems is on overall system throughput not individual transaction latency. This means that no changes are required to individual transactions to take advantage of a parallel execution environment. Instead, the available transactions need only be distributed across the available processors to balance the computational load. This can easily be done within a DSM system.

While our target design environment is a cluster of processors interconnected by a high-speed, switched system area network, the work presented here is also applicable to less tightly-coupled distributed environments such as a conventional network of workstations.

Our focus on object transactions is motivated by their increasing acceptance in the software community, by a need for greater data structuring in transaction processing systems, and as a required enabling technology for the integration of DSM and transactions on cluster based systems.

3 Nested Object Transactions and Nested Object 2PL

We define nested *object* transactions as an extension of Moss' nested transactions [Mos85]. The use of nested object transactions directly supports the automation of consistency maintenance in our system. The start and end of methods define the boundaries of transactions and allow a compiler to insert synchronization operations (lock acquisition and release) automatically. Consistency maintenance is then based on these operations.

3.1 Moss' Nested Transactions

Moss defined nested transactions which permit for the decomposition of monolithic *flat* transactions into subtransactions which may collectively perform the work of the original transactions. Each [sub]-transaction is a sequence of three types of operations: reads and writes of atomic data (the same operations that are provided in conventional, flat transactions) and invocations of sub-transactions. Moss identified the availability of finer granularity concurrency control and recovery as the chief benefits of using nested transactions.

The effect of introducing nested transactions is to induce a tree structure to the processing of data. The top-most transaction (which is user invoked) is referred to as the *root* transaction. Those transactions which do not invoke any sub-transactions are referred to as *leaf* transactions. Between the root and leaf transactions there may be an arbitrary number of *branch* transactions and the exact dimensions of the transaction tree structure are determined by the sub-transaction execution patterns. In Moss' model of nested transactions, data is accessed only by the leaf transactions. Any two sub-transactions which are descendants of the same root transaction are said to be in the same *transaction family* together with it.

Moss divided his nested transactions into open nested transactions and closed nested transactions. The difference between open and closed nested transactions is when updates made by sub-transactions are made visible to other sub-transactions outside the updating transaction's family. With open nested transactions, as soon as a sub-transaction commits, its updates are visible to all other [sub-]transactions regardless of transaction family. This introduces a potential problem should an ancestor of a committed sub-transaction subsequently abort which would require the abortion of all the ancestor's sub-transactions and the associated rollback of their processing. Unfortunately, this is difficult because to undo the effects of any previously committed sub-transaction requires aborting all [sub-]transactions that have accessed data updated by it. This is the classic *cascading aborts* problem. Moss' closed nested transactions avoid this problem by restricting access to updates made by a committed sub-transaction to members of its immediate transaction family (i.e. its parent and siblings and their descendents). In this paper we focus exclusively on such *closed* nested transactions.

3.2 Nested Two Phase Locking (2PL)

Two phase locking (2PL) [BG81, BHG87] is a widely accepted and well understood technique for ensuring the serializability (i.e. correctness) of concurrent transaction executions (distributed or not). Two-phase locking requires each transaction to first acquire all its locks before releasing any. The resulting concurrent execution is provably equivalent [BHG87] to a serial execution of the transactions in the order in which they reach their *lock points*.[2].

Moss extended the concept of two-phase locking to support nested transactions. For closed nested transactions (which are of interest in this paper) the key is a lock *inheritance* mechanism used to enforce the closedness of the transactions' executions. Each sub-transaction acquires locks in the normal way respecting conflicts with other [sub-]transactions. However, there are new rules for lock disposition when a sub-transaction commits or aborts.

When a sub-transaction successfully commits, a process we will refer to as *pre-committing*, its locks are not released for potential acquisition by un-related sub-transactions (possibly from different transaction families). Instead, the locks are *inherited* by its parent transaction which retains them. If a subsequent sub-transaction (direct or indirect) of a retaining [sub-]transaction requires a retained lock, that lock is granted. If any other [sub-]transaction requests the lock, however, it is not granted. In this way, access to updated data that may potentially have to be rolled back later is limited to sub-transactions that will be aborted together with the updating sub-transaction anyway. Thus, no unnecessary transaction roll backs will occur.

When a sub-transaction fails and is forced to abort, its effects are undone and its locks are released unless they were acquired from an ancestor transaction which was retaining them. In this latter case, the locks are again retained by the ancestor transaction. This permits attempted re-execution of the failing sub-transaction so that earlier results need not be unnecessarily discarded.

Only when a root transaction commits are the locks for data updated by its sub-transactions (which are retained by the root) released for acquisition by [sub-]transactions from other transaction families.[3]

[2] A transaction's lock point is defined to be the point at which all its locks have been successfully acquired.

3.3 Nested Object Transactions

In Moss' model all data is both unstructured and unrelated to other data. This is not true when dealing with objects and this represents the fundamental difference between Moss' nested transactions and the nested *object* transactions described in this paper.

In our nested object transactional model, an invocation of a method on an object constitutes a transaction. If such an invocation is made by a user then a root transaction is created. If an invocation is made as part of an existing transaction then the resulting transaction is a sub-transaction of the invoker. This 1:1 mapping between method invocations and transactions produces the expected tree-structured transaction family.

Our nested object transaction model is not constrained to accessing data at leaf transactions. A transaction at any level within a transaction family may access data (via the method code it executes on an object). Further, because of the object structuring, which data will be accessed by a sub-transaction's execution may be predicted (at least conservatively) *a-priori*. This enables a new DSM implementation optimization.

Although it is possible to provide concurrency control at many different levels of granularity, in this paper we choose to use object-level concurrency control. That is, locking occurs on a per-object basis.

3.4 Nested Object Two Phase Locking

As with Moss' nested 2PL protocol, our nested *object* two-phase locking (O2PL) protocol requires a multi-stage lock release process to provide the desired closed nested transaction semantics. The key differences between Moss' nested 2PL and nested O2PL relate to lock acquisition and retention. First, non-leaf transactions must acquire and hold locks for themselves rather than simply retaining them on behalf of their committed sub-transactions. Second, due to the possibility of mutually recursive inter-object invocations, an ancestor transaction may hold (as well as retain) a lock requested by one of its descendants.

The first problem may be solved by a simple extension of Moss' algorithm. The second, however, is more complex. Applying conventional locking rules in the expected way will lead to deadlock within the transaction family involved. This happens because an ancestor transaction will hold a lock that one of its descendants needs. The descendant must wait for the lock to be released before it can execute and the ancestor must wait for all its sub-transactions (including the blocked one) to complete before it can release the lock.

To address this problem, we must prescribe semantics for mutually recursive invocations. At least two alternatives exist. First, we may choose to simply preclude such invocations and verify compliance at run-time (with per-invocation overhead for checking proportional to the depth

[3] Locks that are held by a transaction cannot be acquired by another until they are released. Locks released by non-root transactions are retained by their ancestors. Locks that are retained by a transaction may be acquired but only by a descendant of the retaining transaction.

of transaction nesting at the point of invocation). Alternatively, recursive invocations may be allowed but since the level of concurrency control is per-object, no assumptions can be made about the safety of the resulting invocations. Correctness then becomes the responsibility of the programmer. In this case, locks held by an ancestor may simply be treated in the same way as retained locks for the purpose of acquisition by sub-transactions. In this paper we preclude mutually recursive invocations.

While both these solutions are less than desirable (the first introducing a restriction on programming practice and the latter introducing extra complexity for the programmer) they do not cause us great concern since our experience has been that such mutually recursive invocations are infrequent in practice.

3.5 Automating Nested O2PL

The benefit of using nested object transactions and nested O2PL is that they provide clear demarcation of access boundaries to shared data. Specifically, the statements within an invoked method are the only ones within an overall computation that access the shared data in the given object. This offers the ability to *automatically* insert synchronization primitives (the object lock `acquire` and `release` operations) using the compiler for sub-transactions and the run-time system for root transactions. This permits the use of Entry Consistency (EC) mechanisms for consistency maintenance which minimize consistency message overhead but which require explicit association between shared data and their corresponding synchronization objects. In our system, unlike the unstructured parallel programming environment for which EC was developed, there is no need for programmer involvement in specifying the synchronization objects and operations upon them. The added programming complexity of specifying synchronization operations was the chief criticism of EC. It does not exist in our system.

4 Lazy Object Transactional Entry Consistency

We have developed a simple and efficient page-based DSM consistency protocol for nested object transactions called Lazy Object Transactional Entry Consistency (LOTEC). LOTEC keys on the lock-operations performed by nested O2PL to ensure the consistency of copies of objects cached in different processors' memories with minimal overhead. It also exploits the use of objects for data structuring to optimize consistency maintenance.

4.1 Implementation of LOTEC

Since LOTEC operates based on the locking operations generated for nested O2PL, we begin by explicitly stating the rules governing lock management for nested object transactions:

1. Transaction T may acquire a lock if no other transaction holds a conflicting lock (multiple readers/single writer policy) and all transactions that retain the lock are ancestors of T.

2. Once a lock is acquired by transaction T, the lock is held until T commits or aborts.

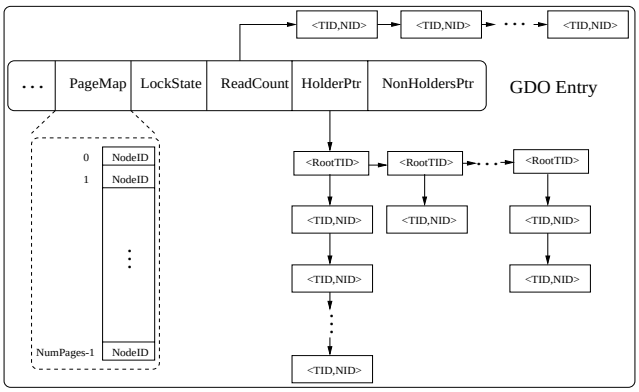

Figure 1: Example GDO Entry Structure (showing lock and consistency fields)

3. A transaction cannot [pre-]commit until all its sub-transactions have pre-committed. When a sub-transaction pre-commits, its parent inherits and retains all of its locks (both held and retained).

4. When a transaction T aborts, it releases all the locks it holds or retains except for those locks retained by any of its ancestors who then continue to retain the locks.

5. When the root transaction commits, it releases all locks it holds or retains thereby making them available to other transaction families.

To facilitate locking operations and consistency maintenance, certain lock structures must be defined. These structures are assumed to be stored in a global directory of objects (GDO) [MGB96] which is uniformly and consistently accessible from all sites. To ensure efficiency and reliability, the GDO design is partitioned and replicated as well as being partially cacheable at local sites.

Each, per object, lock structure contains the following fields (refer to Figure 1):

LockState: A flag indicating the status of the lock (free, held for update, held for read, or retained).

ReadCount: A count of the number of concurrent readers. Incremented when a read lock is acquired and decremented when one is released. Used to determine when shared read access is complete.

HolderPtr: A pointer to a linked list of <transaction_id,node_id> pairs for transactions from the family currently holding the lock who have requested access to the object.

NonHoldersPtr: A pointer to a linked list of lists of <transaction_id,node_id> pairs for transactions from other transaction families who have requested access to the object.

Lock processing is divided into two phases: global operations which access the global GDO information (possibly remotely) and local operations which access locally cached GDO information only. In our current design, individual transaction families execute locally at a single site though competing transactions from different families may access an

object at different sites. This provides a fair system where the cost of executing specific transactions is borne locally at the sites where their root transactions are invoked but the overall load of all transactions is balanced over the available processors in the system. It also leads to a clear division of the lock structure for local and global processing.

The locally cached portion of a GDO entry for a given object consists of the entire list of transactions from the family currently holding the object's lock (`HolderPtr`). This is exactly the information needed to manage the current holding transaction's family's access to the object. Only when a waiting transaction from another family is granted access to the object is communication (of the relevant transaction list) necessary between the GDO and the site executing the transaction family acquiring an object's lock.

In addition to lock information, each GDO entry must also contain information needed by the consistency maintenance algorithm. In the *page-oriented* DSM system we describe in this paper, this information consists of a page map indicating which sites store the most up-to-date versions of each page (refer again to Figure 1). To support this, each site keeps track of which locally cached pages have been made dirty by transaction executions. During the granting of a global lock acquisition request the dirty page information is used to update the GDO page map and to, in part, determine which pages must be propogated to the acquiring site.

Since LOTEC is based on entry consistency, updated object pages are propogated only to the acquiring site (the sole site known to need the updates) at the time of acquisition. This minimizes consistency related network traffic relative to a protocol like release consistency (RC) [CBZ91] which eagerly pushes updates to all caching sites at lock release time. The use of objects to structure shared data permits us to further decrease the communication overhead required for consistency maintenance. EC may still transfer too much data in the case where not all of the updated data will actually be accessed by the acquiring transaction. The use of objects which have well defined access semantics (specified by their methods) permits us to conservatively predict which parts of an object will be accessed by the acquiring transaction. Thus, LOTEC need only transfer those parts of an object (in this system, "pages") which have been updated and which are actually required.

For LOTEC to further reduce the amount of data sent to maintain consistency relative to EC, without added user complexity, it must be possible for the compiler to accurately predict which parts of an object will be accessed by each method. There are two requirements. First, the compiler must be able to detect which attributes will potentially be updated by a given method. This can be easily performed in a conservative[4] fashion by attribute access analysis. In some cases these results can also be improved by the use or partial evaluation techniques. The second feature required of a compiler is to know where, in an object's representation in memory, each attribute is stored. This is a decision which is made by the compiler. Determining which pages will be updated is then simply a matter of mapping attributes to memory pages and, for each method, recording the set of potentially updated pages. The compiler must also annotate the method code by adding a leading call to the local lock acquisition routine and a trailing call to the local lock release routine.

Transferring data in such a *lazy* fashion does introduce some implementation complexity. Specifically, at any given point in time, there may not be a *single* site at which a complete, up-to-date copy of a given object exists. Instead, the up-to-date parts of an object may be scattered throughout the distributed system on multiple nodes. The locations of the up-to-date pages of each object are tracked in the GDO using the page map described earlier. Fortunately, this information can be maintained with low overhead. Dirty page information may be piggybacked on each global lock release message and a site map containing the locations of the most up-to-date object pages may be sent during global lock acquisition to the acquiring site's transaction. This ensures that the bulk of processing is performed locally.

Sketches of algorithms describing the processing performed during local and global lock acquisition, during local and global lock release, and for the transfer of updated pages are now presented. These algorithms illustrate the interaction between the locally and remotely executed operations required for lock management. Note that a minimum of network traffic is required in order to effect locking operations.

Algorithm 4.1 *LocalLockAcquisition*

```
IF the object is not cached at this site THEN
    Forward request to GlobalLockAcquisition
ELSE
    IF the requesting transaction belongs to the current
    holder's family THEN
        IF the lock is retained by an ancestor of the re-
        quester THEN
            Grant the lock (R or W) to the transaction
        ELSE /* currently locked by another transaction
        in the family */
            IF request is for a Write or the lock is held
            for Writing THEN
                Link transaction onto local list
            ELSE
                Grant the Read lock to the requesting
                transaction
    ELSE /* held or retained by another transaction family
    executing at this site */
        Forward request to GlobalLockAcquisition
```

End of Algorithm

Algorithm 4.2 *GlobalLockAcquisition*

```
Lookup appropriate entry in GDO
IF the lock is free THEN
    Set the lock to 'held' and update the HolderPtr to point
    to the requesting transaction
    Send the list pointed to by HolderPtr and the object's
    page map to the requesting transaction's site
ELSE IF the lock is held for Read and this is a Read request
THEN /* concurrent reading is OK */
    Grant the Read lock to the requesting transaction
ELSE /* lock is not free or conflicting request */
    IF there is a list pointed to by NonHoldersPtr for the
    requesting transaction's family THEN
        Link the requesting transaction into its family's
        list
    ELSE
        Create a new list for the requester's family and
        link into the list pointed to by NonHoldersPtr
```

End of Algorithm

[4]The run-time value of data may alter control flow within a method. Thus, it is not possible to exactly predict which data will be accessed because the control path(s) followed cannot be known. *Conservative* predictions are made so that regardless of which of the possible paths are taken, and, thus, which attributes are accessed, *all* possibly updated attributes will be recorded.

Lock release processing is somewhat different than acquisition processing in that it potentially deals with multiple objects. The 'LocalLockRelease' routine is invoked when a transaction commits or aborts. Since such a transaction may have executed sub-transactions, release processing must deal with the disposition of those locks.

Algorithm 4.3 *LocalLockRelease*

```
CASE sub-transaction pre-commits
    FOREACH object updated /* by a sub-transaction execution */ DO
        Update dirty page map
        Release lock to parent transaction for retaining
        IF there is a sub-transaction or the retainer waiting THEN
            Grant the lock to that sub-transaction
CASE root transaction commits
    Forward request to GlobalLockRelease /* with dirty page info */
CASE root transaction aborts
    UNDO updates
    Forward request to GlobalLockRelease /* no dirty page info */
CASE sub-transaction aborts
    UNDO updates
    FOREACH object updated /* by a sub-transaction execution */ DO
        IF the object is retained by an ancestor THEN
            The ancestor again retains the lock
            IF there is a sub-transaction or the retainer waiting THEN
                Grant the lock to that sub-transaction
        ELSE /* not retained by an ancestor */
            Forward request to GlobalLockRelease /* no dirty page info */
```

End of Algorithm

Note that the UNDO operations required by the 'LocalLockRelease' routine may be done using either *local* UNDO logs or shadow pages. In either case, no network communication is required.

Algorithm 4.4 *GlobalLockRelease*

```
FOREACH object updated DO /* corresponds release due to commits not aborts */
    Record the NodeIdentifier of the updating site in the GDO for each updated page (if update took place, could be Read access)
FOREACH lock being released DO
    IF no other transaction is waiting for the lock THEN
        Set LockState to 'Free' and HolderPtr to NULL
    ELSE
        Unlink the next transaction list from NonHoldersPtr and link onto HolderPtr
        Send the list pointed to by HolderPtr and the page map to the new holder's site
```

End of Algorithm

Algorithm 4.5 *TransferOfUpdatedPages*

```
FOREACH object page DO
    IF the most up-to-date page is not resident here THEN
        Add the page to a list of pages to obtain from the site at which it is stored
FOREACH site from which page(s) must be obtained DO
    Copy the set of pages provided in the site's list from the specified site to the acquiring site
```

End of Algorithm

The processing performed by the 'TransferOfUpdatedPages' routine is carried out by the site running the transaction which has just acquired the relevant object's lock.

4.2 False Sharing

The issue of false sharing was not explicitly addressed in the algorithms just presented. This is because false sharing can be implicitly handled in LOTEC. LOTEC is object based and concurrency control and consistency maintenance is provided at the object level not the page level. The extent (address range in memory) of an object is easily known. Further, changes made to an object residing on a given page may be freely updated on other nodes in the distributed system without affecting the state of other objects that happen to be stored on the same page. Although LOTEC is described as being a page-based DSM system in this paper, only updates to the objects (not the entire pages they are stored on) really need to be transmitted between nodes. In this respect, LOTEC is more like a Distributed Shared Data system. This handling of false sharing means the use of diffs (as in [KCZ92]) and their associated overhead (i.e. page twinning and diff computation) is unnecessary.

4.3 Correctness of LOTEC

We now provide a sketch of the reasoning behind a correctness proof for LOTEC. There are two aspects to correctness which must be considered; the correctness of nested O2PL, and the correctness of LOTEC itself. The correctness of nested O2PL will guarantee that no invalid results will occur as a result of concurrency between transactions. The correctness of LOTEC will guarantee that any distributed execution of transactions which would execute correctly under O2PL on a uniprocessor will still be correct.

To illustrate the correctness of nested O2PL, we depend on the proven correctness of nested 2PL [BHG87]. For nested O2PL to ensure correct concurrent executions, it must ensure that the transactions it schedules are serializable. Locking occurs at the object level and each method invocation constitutes a transaction. Conflicting operations on *different* objects are serializable because such operations are scheduled according to the existing nested 2PL locking rules. This scheduling is implemented using the lock inheritance mechanism described earlier. In nested O2PL it is also possible to have concurrent operations on a single object but only within a single transaction family. As discussed earlier, such concurrency is either forbidden or its correctness is left to the programmer.

For a distributed execution of transactions which would execute correctly under O2PL on a uniprocessor to also be correct, a transaction running on a given node must always access the most up to date (as defined by O2PL) version of each object. By the use of O2PL, we know only a single transaction, running on a specific node, will ever access its local copy of the object at a time. We can easily ensure that the most up to date version of an object is always available at all nodes by simply copying the updated local copy to all other nodes before it can be accessed there. This can safely be done between local lock release and a subsequent global

lock acquisition. This is essentially what Release Consistency (RC) does and it is known to be correct [CBZ91]. To decrease the number and volume of consistency messages sent, LOTEC optimizes this processing in two ways. First, it delays the transfer of updates until global lock acquisition at which time data transfer need only be done between the node which last updated the object and the node running the acquiring transaction. This is what is done in Entry Consistency [BZS93]. Second, it transfers only those parts of the object which are predicted to be needed. If additional parts turn out to be needed, these can be fetched on demand. Finally, since not all needed parts of an object necessarily reside at the node where the object was last updated it may be necessary to collect parts from several nodes. This is accomplished using the data structures and algorithms described previously.

For further details on the correctness of the LOTEC protocol, please refer to Sui's thesis [Sui98].

5 Simulation of LOTEC Algorithms

As a first step towards the implementation of our DSM based persistent object system and to increase our confidence in the correctness of our algorithm implementations, we constructed a simulation/emulation system. This system included a simple GDO data structure and implementations of the algorithms described in this paper which were used to manage concurrency between, and ensure consistency for, a number of randomly generated nested object transactions in a simulated distributed system. The simulation experiments support the correctness of our implementations.

While there is no system we know of that is appropriate for use in a direct performance comparison with LOTEC, we felt it was important to assess the effectiveness of some of the design decisions we made in creating LOTEC. In particular, we were interested in comparing the performance offered by the use of entry consistency alone versus that provided when only the predicted "required" pages were transferred (a novel feature of LOTEC). To do this we created a suite of three consistency protocols and simulated them all. The simplest of these was called COTEC (for Conservative Object Transactional Entry Consistency). COTEC transfers *all* of an object's pages to the acquiring site after a successful lock acquisition and provides a baseline for performance measurement. The second protocol was referred to as OTEC and optimized COTEC by sending only the updated pages to an acquiring transaction's site. The third protocol was LOTEC, as described in this paper which sends only those updated pages which are predicted to be needed. The simulation was expressly designed to induce high degrees of conflict in object access as this is the interesting case. We naturally expect performance to be good in low-conflict situations as well.

We varied the number of objects, the size of the objects (in units of pages) and the number of transactions in order to achieve a range of conflict scenarios. As expected, with fairly heavy degrees of conflict in object access and complex objects consisting of multiple pages (only a subset of which are normally updated by any method/transaction), LOTEC outperforms OTEC which outperforms COTEC. While the performance of the three protocols varies by scenario, OTEC generally outperforms COTEC by approximately 20 - 25% while LOTEC outperforms OTEC by another 5 - 10%. In some cases, the difference is more dramatic but with a synthetic workload of transactions we do not want to speculate on the importance of these results.

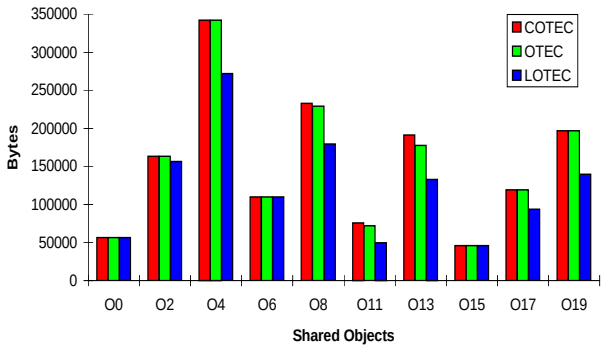

Figure 2: Medium Sized Objects with High Contention

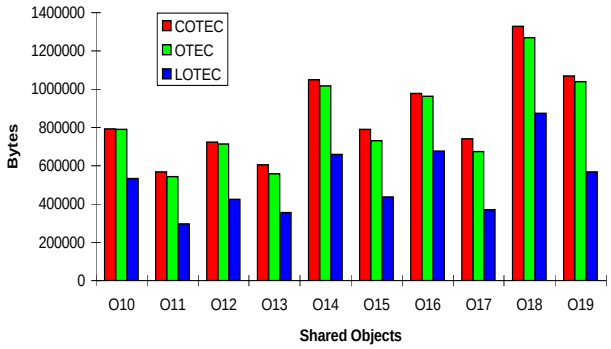

Figure 3: Large Sized Objects with High Contention

Graphs showing the number of bytes transferred by each of the three protocols to maintain the consistency of various shared objects are shown in Figures 2 through 5. The objects were selected to reflect a variety of reference patterns that arose in the randomized nested transactions. In Figure 2 and Figure 4 results for medium sized objects (on the order of one to five pages) are shown under high and moderate contention respectively. In Figure 3 and Figure 5 results for larger objects of ten to twenty pages are similarly shown.

These simulation results show that, as expected, LOTEC sends the smallest amount of data in order to maintain consistency. We realized, however, that this was done at a particular cost. LOTEC also sends many *more* messages (albeit small ones) than OTEC or COTEC. This suggested the importance of low message latency for LOTEC. We instrumented our simulator to assess the effects of changing the network bandwidth and message initiation overhead to determine the range of network parameters for which the use of LOTEC would make sense.

We simulated the various protocols at bit rates roughly corresponding to *switched* (i.e. no collisions) conventional, fast, and gigabit Ethernet. Despite sending more messages in some cases, LOTEC faired quite well for the slower networks even with fairly heavyweight messaging protocols. As the speed of the network increased though the impor-

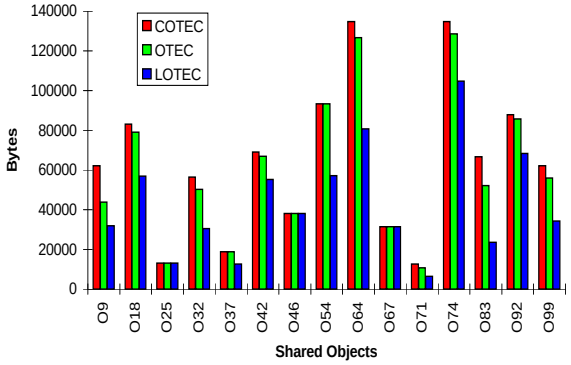

Figure 4: Medium Sized Objects with Moderate Contention

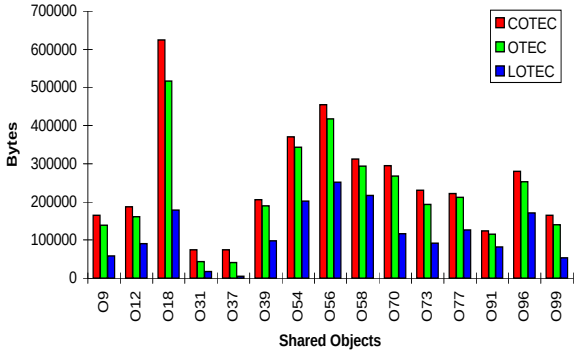

Figure 5: Large Sized Objects with Moderate Contention

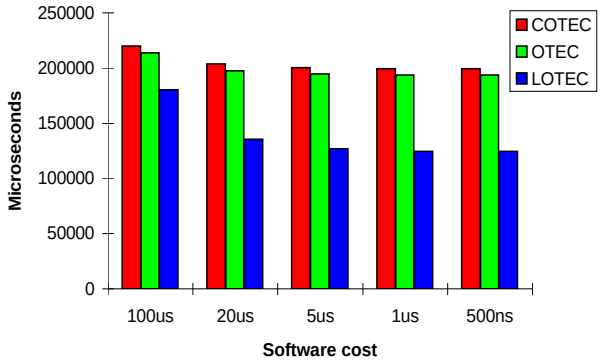

Figure 6: Example Transfer Time at 10Mbps

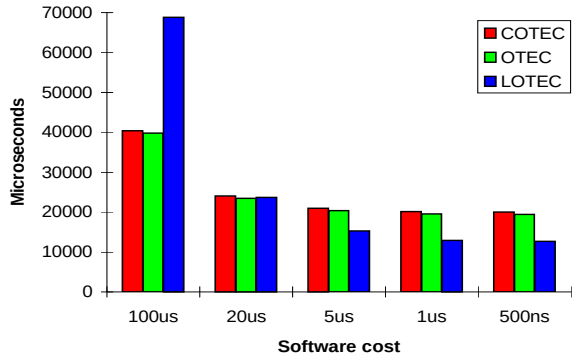

Figure 7: Example Transfer Time at 100Mbps

tance of low software messaging overhead became critical to LOTECs performance. Our experiments allow us to conclude that LOTEC should perform well with current, fast Ethernet networks using only mildy aggressive, low-latency network protocols. As we migrate to gigabit Ethernet though any LOTEC implementation will also have to incorporate extremely efficient message transmission protocols.

Figures 6 through 8 show the total message time required to maintain the consistency of an arbitrary shared object throughout the simulation experiments. Times are reported for 10Mbps, 100Mbps, and 1Gbps networks respectively. Further, for each network type, the total message time is provided for a number of possible message startup latencies. This allows us to assess the potential messaging performance of LOTEC under any of a number of schemes that reduce the *software*-induced factors in message latency. The importance of minimizing message latency has recently been illustrated by the success of the Millipage system [IS99] which exploits Gbps networks and low-latency protocols to support fine-granularity *sequential* consistency with performance comparable to existing weak consistency systems.

5.1 Locking Overhead

The overall performance of a distributed object system built using the LOTEC protocol will be determined not just by the cost of ensuring consistency but also by other factors.

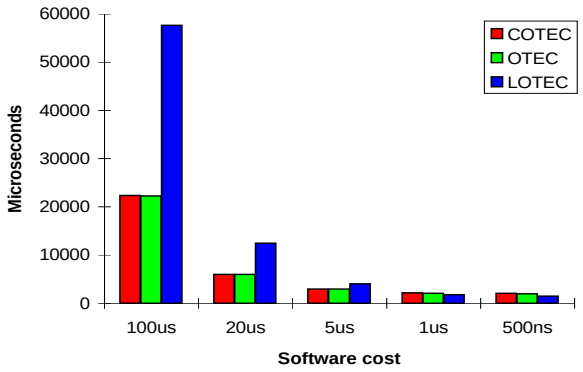

Figure 8: Example Transfer Time at 1Gbps

Among these is the overhead associated with locking operations.

Each lock acquisition performed at a site other than where the corresponding object was last updated will require a message to the GDO. While such messages are small, the time required to send each one and receive a reply is typically much greater than the time required to perform a local operation. Keeping the overhead of lock operations small is an important implementation issue that can be addressed in a number of ways.

The LOTEC protocol, as described, has a natural preference for coarse-grained concurrency since the larger objects are, the fewer lock operations are necessary. Not all object systems, however, exhibit coarse granularity. Heavily object-based environments can sometimes aggregate *related* small objects into larger objects for the purpose of decreasing the cost of concurrency control and consistency maintenance. While this is not optimal for all applications, it is appropriate for many where the smallest objects are commonly elements of larger structures. [5] Our focus on coarse-grained objects was reflected in the simulation experiments which addressed only comparatively large, multi-page objects.

Support for fine-granularity objects must be provided using other techniques. In addition to low-latency network protocols, we are investigating asynchronous messaging strategies for distributed lock maintenance. We are also actively pursuing aggressive access prediction techniques for object management. Just as we can conservatively predict which parts of an object a method may access, we can also predict which other objects a given method may invoke methods on. This information can then be used to permit optimistic pre-acquisition of locks in the GDO as well as pre-fetching of needed objects and their GDO entries. Performing these operations in parallel with other operations effectively hides the latency of remote lock acquisition thereby improving overall performance.

It must be remembered that the overhead of synchronization operations is a cost associated with the benefits of transactional execution. Similarly, the overhead of messaging is a necessary cost to gain the benefits of distribution. It would clearly not be fair to compare the performance of LOTEC to that of a non-distributed object system lacking any form of concurrency control. Finally, it should also be noted that the overhead of locking operations may be offset by other performance improvements offered by LOTEC (e.g. decreased overhead since no additional processing is needed to handle false sharing).

[5] This includes computer aided design environments for which this work was originally developed.

6 Conclusions and Future Work

The chief contributions of this paper are:

1. the application of DSM to a new area of high performance computation (transactional object systems),
2. showing the compatibility of DSM techniques and nested object transactions, and
3. description of a new DSM consistency protocol, LOTEC, which exploits object semantics to further decrease the amount of data (relative to EC) that must be sent to maintain consistency and which obviates the need for explicit management of false sharing.

One omission from our simulation studies was the implementation of a simulated version of Release Consistency for nested objects. This work is now underway and will allow us to compare the results of using that protocol to the results offered by COTEC, OTEC and LOTEC. We are also actively expanding our simulation system to verify LOTEC's compatibility with conventional DSM optimization techniques including the use of multicast-capable networks, and scope consistency.

Future research will include an exploration of extensions to support different consistency protocols and recovery mechanisms on a per-class basis, application of LOTEC to distributed shared data (DSD) rather than distributed shared memory (DSM) systems, and the integration of active messaging into LOTEC to improve its performance for gigabit networks.

Finally, an actual implementation in the form of a Linux kernel module is now underway.

References

[ACDZ97] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, pages 261–271, February 1997.

[ACRZ97] C. Amza, A. L. Cox, K. Ramajamni, and W. Zwaenepoel. Tradeoffs between False Sharing and Aggregation in Software Distributed Shared Memory. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 90–99, June 1997.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles J. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet - A Gigabit-per-second Local-Area Network. *IEEE Micro*, 15(1), February 1995.

[BG81] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185 – 221, 1981.

[BHAB96] S. Ben Hassen, I. Athanasiu, and H.E. Bal. A Flexible Operation Execution Model for Shared Distributed Objects. In *Proc. of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '96)*, October 1996.

[BHG87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[BPA98] R. Bianchini, R. Pinto, and C. L. Amorim. Data Prefetching for Software DSMs. In *Proc. of the Int'l Conf. on Supercomputing'98*, July 1998.

[BTSD+95] Donald J. Becker, Daniel Savarese Thomas Sterling, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proc. of the 1995 Int'l Conf. on Parallel Processing (ICPP'95)*, April 1995.

[Buo98] Philip Buonadonna. Implementation and Analysis of the Virtual Interface Architecture. In ACM, editor, *Proceedings of the 1998 ACM/ IEEE SC98 Conference, Orlando, Florida, USA, November 7–13, 1998*, 1998.

[BZS93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. In *Proc. of the 38th IEEE Int'l Computer Conf. (COMPCON Spring'93)*, pages 528–537, February 1993.

[CBZ91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, October 1991.

[CBZ95] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Trans. on Computer Systems*, 13(3):205–243, August 1995.

[Deu90] O. Deux *et al.* The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91 – 108, 1990.

[DVL92] Simon H. D., Dalsem W. R. V., and Dagum L. *Parallel CFD: Current Status and Future Requirements*. The MIT Press, 1992.

[FH98] B. D. Fleisch and R. L. Hyde. High Performance Distributed Objects Using Distributed Shared Memory and Remote Method Invocation. In *Proc. of the 31st Hawaii Int'l Conf. on System Sciences (HICSS-31)*, volume VII, pages 574–578, January 1998.

[IS99] Ayal Itzkovitz and Assaf Schuster. MultiView and Millipage–Fine-Grain Sharing in Page-Based DSMs. In *Proc. of the Conference on Operating Systems Desin and Implementation (OSDI '99)*, February 1999.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[LH86] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proc. of the 5th Annual ACM Symp. on Principles of Distributed Computing (PODC'86)*, pages 229–239, August 1986.

[LLOW91] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. The ObjectStore System. *Communications of the ACM*, 34(10):50–63, October 1991.

[Lu97] Paul Lu. Aurora: Scoped Behaviour for Per-Context Optimized Distributed Data Sharing. In *Proc. of the 11th Int'l Parallel Processing Symp. (IPPS'97)*, April 1997.

[MGB96] John Mathew, Peter Graham, and Ken Barker. Object Directory Design for a Fully Distributed Persistent Object System. In *Proceedings of the Engineering Systems Design and Analysis Conference (ESDA'96)*, Montpellier, France, July 1996.

[Mos85] J.E.B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing*. The MIT Press, 1985.

[RB94] T. Ruhl and H. E. Bal. The Nested Object Model. In *Proc. of the 6th ACM SIGOPS European Workshop*, September 1994.

[SB98] W. E. Speight and J. K. Bennett. Reducing Coherence-Related Communication in Software Distributed Shared Memory Systems. Technical Report ECE TR-98-03, Electrical and Computer Engineering Department, Rice University, 1998.

[SF95] M. Shapiro and P. Ferriera. Larchant-RDOSS: A Distributed Shared Persistent Memory and Its Garbage Collector. In *Proc. of the 9th Int'l Workshop on Distributed Algorithms (WDAG'95)*, pages 198–214, September 1995.

[Sui98] Y. Sui. DSVM Consistency Protocols for Nested Object Transactions. Master's thesis, Dept. of Computer Science, University of Manitoba, August 1998.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 303–316, December 1995.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–267, Gold Coast, Australia, May 1992.